作业一：使用 python 语言模拟实现银行家算法，要求封装成一个函数，能够接收 Max、Need、Available、Allocated 矩阵，以及资源申请 Request， 使用银行家算法计算后输出是否能够分配，以及分配后的四个矩阵。

解答：

以下是 python 语言代码模拟实现银行家算法：

```
import numpy as np   #引入 numpy 库，用于矩阵/向量的便捷处理

#定义银行家算法函数
def bankers_algorithm(Max,Allocated,Available, Request, process_id):

#将输入的列表转换为 numpy 数组，便于后续向量/矩阵操作
    Max = np.array(Max)
    Allocated = np.array(Allocated)
    Available = np.array(Available)
    Request = np.array(Request)

#根据 Max 和 Allocated 计算出每个进程当前还需要多少资源
    Need = Max - Allocated

# Step 1：检查请求是否合法
    if any(Request > Need[process_id]):
```

#若请求超过了该进程的最大剩余需求，非法，拒绝

return False, "Error: 请求超过需求", None

if any(Request > Available):

#若请求超过了当前系统的可用资源，无法满足，拒绝

return False, "Error: 请求超过可用资源", None

#Step 2：试探性分配资源（临时修改资源状态）

Available_temp = Available - Request #假设分配后可用资源减少

Allocated_temp = Allocated.copy() #拷贝当前分配表

Allocated_temp[process_id] += Request #给请求进程分配资源

Need_temp = Max - Allocated_temp #更新对应的 Need 矩阵

#Step 3：进行安全性检查

work = Available_temp.copy() #当前可用资源，用于模拟“资源回收”

finish = [False] * len(Max) #标记每个进程是否能顺利完成

safety_sequence = [] #存放安全序列（完成顺序）

print("\n======= 安全性检查过程输出 =======")

#安全性检测主循环：尝试找出所有可以“完成”的进程

while True:

allocated_this_round = False #本轮是否至少有一个进程完

成

```
        for i in range(len(Max)):
            #条件：进程 i 尚未完成，且它的 Need[i]≤当前 Work，可完成
            if not finish[i] and all(Need_temp[i] <= work):
                #打印当前进程状态（调试/展示用）
                print(f"\nProcess {i}:")
                print(f"    Work: {work}")
                print(f"    Need: {Need_temp[i]}")
                print(f"    Allocation: {Allocated_temp[i]}")
                print(f"        Work + Allocation: {work + Allocated_temp[i]}")

                work += Allocated_temp[i]    #模拟释放资源
                finish[i] = True             #该进程标记为完成
                safety_sequence.append(i)    #加入安全序列
                allocated_this_round = True
        if not allocated_this_round:
            #若没有进程可以完成，跳出循环（可能陷入不安全）
            break

    #判断是否所有进程都能顺利完成
```

```
    if all(finish):
        print("\nSafety Sequence:", safety_sequence) #打印安全序列
        print("Allocate Successfully")          #输出成功分配提示
        return True, "资源可以安全分配", {
            "Available": Available_temp.tolist(), #返回分配后的系统资源状态
            "Allocated": Allocated_temp.tolist(),
            "Need": Need_temp.tolist(),
            "Max": Max.tolist()
        }
    else:
        print("\n 系统进入不安全状态，无法完成所有进程")  # 输出失败原因
        return False, "系统进入不安全状态，分配失败", None


#主函数部分：用于运行测试用例
if __name__ == "__main__":
    #最大资源需求矩阵：每个进程最多可能需要的资源数
    Max = [[7, 5, 3],
           [3, 2, 2],
```

[9, 0, 2],

[2, 2, 2],

[4, 3, 3]]

#当前已分配资源矩阵：每个进程当前持有的资源数

Allocated = [[0, 1, 0],

[2, 0, 0],

[3, 0, 2],

[2, 1, 1],

[0, 0, 2]]

#当前系统中可用的资源数

Available = [3, 3, 2]

#进程 P1 发出的资源请求向量

Request = [1, 0, 2]

process_id = 1 # 请求来自进程编号为 1 的进程（P1）

#调用银行家算法函数

result, message, new_matrices = bankers_algorithm(Max, Allocated, Available, Request, process_id)

```
    #打印请求判断结果
    print("\n 请求结果:", message)
    if result:
        print("\n======= 分配后资源状态 =======")
        #若分配成功，输出更新后的四个关键资源状态矩阵
        for key, val in new_matrices.items():
            print(f"{key}:\n{np.array(val)}")
```

在运行上面 python 代码之后，得到如下结果：

======= 安全性检查过程输出 =======

Process 1:

Work: [2 3 0]

Need: [0 2 0]

Allocation: [3 0 2]

Work + Allocation: [5 3 2]

Process 3:

Work: [5 3 2]

Need: [0 1 1]

Allocation: [2 1 1]

Work + Allocation: [7 4 3]

Process 4:

Work: [7 4 3]

Need: [4 3 1]

Allocation: [0 0 2]

Work + Allocation: [7 4 5]

Process 0:

Work: [7 4 5]

Need: [7 4 3]

Allocation: [0 1 0]

Work + Allocation: [7 5 5]

Process 2:

Work: [7 5 5]

Need: [6 0 0]

Allocation: [3 0 2]

Work + Allocation: [10 5 7]

Safety Sequence: [1, 3, 4, 0, 2]

Allocate Successfully

请求结果：资源可以安全分配

======= 分配后资源状态 =======

Available:

[2 3 0]

Allocated:

[[0 1 0]

[3 0 2]

[3 0 2]

[2 1 1]

[0 0 2]]

Need:

[[7 4 3]

[0 2 0]

[6 0 0]

[0 1 1]

[4 3 1]]

Max:

[[7 5 3]

[3 2 2]

[9 0 2]

[2 2 2]

[4 3 3]]

对于上述结果，解释如下：

首先，请求合法性判断。进程 P1 请求资源向量[1, 0, 2]，首先进行以下两个判断：

第一，请求是否超过 Need[P1]？

Need[P1] = Max[P1] - Allocated[P1] = [3,2,2] - [2,0,0] = [1,2,2]

请求 [1,0,2] ≤ Need[P1]合法

第二，请求是否超过 Available？

当前 Available = [3,3,2]

请求[1,0,2] ≤ Available 合法

由以上可知，请求合法，进入试探性资源分配阶段。

其次，试探性分配后系统状态。模拟将请求分配给 P1 后，更新如下：

Available = [3,3,2] - [1,0,2] = [2,3,0]

Allocated[P1] = [2,0,0] + [1,0,2] = [3,0,2]

Need[P1] = [1,2,2] - [1,0,2] = [0,2,0]

然后，安全性检查过程输出（主循环）。系统将尝试寻找一个安全序列，使所有进程能依次完成并释放资源。过程如下：

Process 1:

Work = [2 3 0]（初始 Available）

Need[1] = [0 2 0] ≤ Work

进程 P1 完成，释放资源：

Work + Allocation[1] = [2 3 0] + [3 0 2] = [5 3 2]

Process 3:

Work = [5 3 2]

Need[3] = [0 1 1] ≤ Work

P3 完成，释放资源：

Work + Allocation[3] = [5 3 2] + [2 1 1] = [7 4 3]

Process 4:

Work = [7 4 3]

Need[4] = [4 3 1] ≤ Work

P4 完成，释放资源：

Work + Allocation[4] = [7 4 3] + [0 0 2] = [7 4 5]

Process 0:

Work = [7 4 5]

Need[0] = [7 4 3] ≤ Work

P0 完成，释放资源：

Work + Allocation[0] = [7 4 5] + [0 1 0] = [7 5 5]

Process 2:

Work = [7 5 5]

Need[2] = [6 0 0] ≤ Work

P2 完成，释放资源：

Work + Allocation[2] = [7 5 5] + [3 0 2] = [10 5 7]

所有进程都可以顺利完成，安全序列如下：

Safety Sequence: [1, 3, 4, 0, 2]

Allocate Successfully

即 P1 → P3 → P4 → P0 → P2，是本次资源分配操作下的合法安全序列。

最后，分配后系统状态输出。分配成功后，系统的四个关键矩阵更新如下：

1. Available：当前系统可分配资源

[2 3 0]

表示系统尚剩余：2 个 A 类、3 个 B 类、0 个 C 类资源。

2. Allocated：各进程当前占有资源

[[0 1 0]

[3 0 2]

[3 0 2]

[2 1 1]

[0 0 2]]

可以看到，P1 从[2, 0, 0]增加到[3, 0, 2]。

3. Need：各进程剩余需求（Max - Allocated）

[[7 4 3]

[0 2 0]

[6 0 0]

[0 1 1]

[4 3 1]]

P1 的 Need 更新为[0, 2, 0]，说明它还需要 2 个 B 资源即可完成任务。

4. Max：最大资源需求（初始不变）

[[7 5 3]

[3 2 2]

[9 0 2]

[2 2 2]

[4 3 3]]

由以上结果，我们验证了银行家算法在处理进程资源请求时能够有效防止系统进入不安全状态。该算法通过，判断请求合法性；试探性资源分配；安全性检查（构建安全序列）来确保系统在任意时刻都能维持整体安全。此次进程 P1 的请求[1, 0, 2]被系统接受，更新后系统仍存在安全序列[1, 3, 4, 0, 2]，说明分配是成功且安全的。

但是，有时候即使请求合法（满足 Request ≤ Need 且 Request ≤ Available），但是如果分配后，系统无法保证所有进程都能顺利执行完释放资源，银行家算法就会判断为“不安全”，拒绝分配。我们运行以下 python 代码：

```
import numpy as np
def bankers_algorithm(Max, Allocated, Available, Request, process_id):
    Max = np.array(Max)
    Allocated = np.array(Allocated)
    Available = np.array(Available)
    Request = np.array(Request)
```

```
Need = Max - Allocated
if any(Request > Need[process_id]):
    return False, "Error: 请求超过需求", None
if any(Request > Available):
    return False, "Error: 请求超过可用资源", None
Available_temp = Available - Request
Allocated_temp = Allocated.copy()
Allocated_temp[process_id] += Request
Need_temp = Max - Allocated_temp
work = Available_temp.copy()
finish = [False] * len(Max)
safety_sequence = []
print("\n=======  安全性检查过程输出  =======")
while True:
    allocated_this_round = False
    for i in range(len(Max)):
        if not finish[i] and all(Need_temp[i] <= work):
            print(f"\nProcess {i}:")
            print(f"    Work: {work}")
            print(f"    Need: {Need_temp[i]}")
            print(f"    Allocation: {Allocated_temp[i]}")
            print(f"        Work    +    Allocation:    {work    +
```

```
Allocated_temp[i]}")
                    work += Allocated_temp[i]
                    finish[i] = True
                    allocated_this_round = True
                    safety_sequence.append(i)
            if not allocated_this_round:
                break
        if all(finish):
            print("\nSafety Sequence:", safety_sequence)
            print("Allocate Successfully")
            return True, "资源可以安全分配", {
                "Available": Available_temp.tolist(),
                "Allocated": Allocated_temp.tolist(),
                "Need": Need_temp.tolist(),
                "Max": Max.tolist()
            }
        else:
            print("\n 系统进入不安全状态，无法完成所有进程")
            return False, "系统进入不安全状态，分配失败", None
if __name__ == "__main__":
    Max = [[3, 2, 2],
           [6, 1, 3],
```

```
            [3, 1, 4],
            [4, 2, 2],
            [5, 3, 3]]
    Allocated = [[1, 0, 0],
                 [6, 1, 2],
                 [2, 1, 1],
                 [0, 0, 2],
                 [0, 0, 2]]
    Available = [0, 0, 0]   # 没有可用资源，系统处于危险边缘
    Request = [0, 0, 1]      # 进程 0 请求 1 个资源（合法）
    process_id = 0
    result, message, new_matrices = bankers_algorithm(Max,
Allocated, Available, Request, process_id)
    print("\n 请求结果:", message)
    if result:
        print("\n======= 分配后资源状态 =======")
        for key, val in new_matrices.items():
            print(f"{key}:\n{np.array(val)}")
```

得到如下结果：

请求结果: Error: 请求超过可用资源。

该结果表明，在本次实验中，虽然进程 P0 请求的资源在形式上是合法的（未超过其最大需求），但由于系统当前可用资源为[0, 0, 0]，不

足以满足其请求[0, 0, 1]，银行家算法在第一阶段即判断请求超出系统能力范围，立即返回错误提示并拒绝资源分配。这体现了算法在资源管理上的前置防御机制，优先保障系统稳定性和安全性。

我们再给出一个“合法但不安全”的反例，python 代码如下：

```
import numpy as np
def bankers_algorithm(Max, Allocated, Available, Request, process_id):
    Max = np.array(Max)
    Allocated = np.array(Allocated)
    Available = np.array(Available)
    Request = np.array(Request)
    Need = Max - Allocated
    # Step 1: 请求合法性检查
    if any(Request > Need[process_id]):
        return False, "Error: 请求超过需求", None
    if any(Request > Available):
        return False, "Error: 请求超过可用资源", None

    # Step 2: 试探性分配
    Available_temp = Available - Request
    Allocated_temp = Allocated.copy()
    Allocated_temp[process_id] += Request
```

```
    Need_temp = Max - Allocated_temp
    work = Available_temp.copy()
    finish = [False] * len(Max)
    safety_sequence = []
    print("\n=======  安全性检查过程输出  =======")
    while True:
        allocated_this_round = False
        for i in range(len(Max)):
            if not finish[i] and all(Need_temp[i] <= work):
                print(f"\nProcess {i}:")
                print(f"    Work: {work}")
                print(f"    Need: {Need_temp[i]}")
                print(f"    Allocation: {Allocated_temp[i]}")
                print(f"        Work + Allocation: {work +
Allocated_temp[i]}")
                work += Allocated_temp[i]
                finish[i] = True
                safety_sequence.append(i)
                allocated_this_round = True
        if not allocated_this_round:
            break
    if all(finish):
```

```
            print("\nSafety Sequence:", safety_sequence)
            print("Allocate Successfully")
            return True, "资源可以安全分配", {
                "Available": Available_temp.tolist(),
                "Allocated": Allocated_temp.tolist(),
                "Need": Need_temp.tolist(),
                "Max": Max.tolist()
            }
        else:
            incomplete = [i for i, done in enumerate(finish) if not done]
            print("\n 系统进入不安全状态，无法完成所有进程")
            print("未完成进程:", incomplete)
            return False, "系统进入不安全状态，分配失败", None


# 示例输入（合法但不安全）
if __name__ == "__main__":
    Max = [
        [3, 2, 2],
        [6, 1, 3],
        [3, 1, 4],
        [4, 2, 2],
```

```
        [5, 3, 3]
    ]

    Allocated = [
        [1, 0, 0],
        [6, 1, 2],
        [2, 1, 1],
        [0, 0, 2],
        [0, 0, 2]
    ]

    Available = [0, 0, 1]    #系统资源紧张

    Request = [0, 0, 1]    #进程 0 合法请求
    process_id = 0

    result,message,new_matrices=bankers_algorithm(Max,
Allocated, Available, Request, process_id)
    print("\n 请求结果:", message)
    if result:
        print("\n=======  分配后资源状态  =======")
        for key, val in new_matrices.items():
```

print(f"{key}:\n{np.array(val)}")

代码运行结果如下：

======= 安全性检查过程输出 =======

系统进入不安全状态，无法完成所有进程

未完成进程: [0, 1, 2, 3, 4]

请求结果: 系统进入不安全状态，分配失败

由以上结果我们可知，进程 P0 提出了一个合法的资源请求[0, 0, 1]，该请求未超过其最大需求（Need）且小于等于系统当前可用资源（Available = [0, 0, 1]），因此从请求合法性判断来看，满足银行家算法的第一步条件。系统随后进入安全性检查阶段，试探性地将资源分配给进程 P0，并更新系统资源状态。通过模拟发现，系统无法找到一个使所有进程都能依次完成的安全执行序列。输出结果中明确显示：

系统进入不安全状态，无法完成所有进程

未完成进程: [0, 1, 2, 3, 4]

这表明没有任何一个进程能在当前资源状态下顺利完成并释放资源给其他进程，系统将陷入可能的死锁。因此，虽然资源请求是合法的，但银行家算法为了保证系统安全，拒绝此次请求，避免进入不安全状态。这正是银行家算法的核心机制：宁可不分配，也不冒险死锁。